CENG 3420
Computer Organization & Design

Lecture 05: Arithmetic and Logic Unit – 1

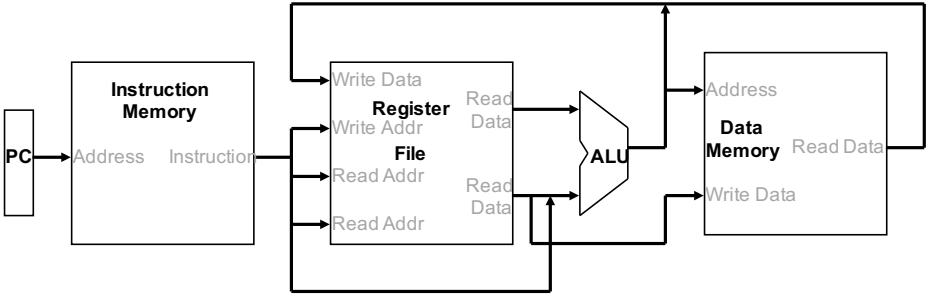
Bei Yu
CSE Department, CUHK
byu@cse.cuhk.edu.hk

(Latest update: January 20, 2022)

Spring 2022

Overview

Abstract Implementation View



Arithmetic

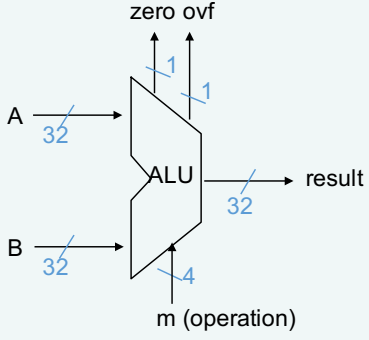
Where we've been: **abstractions**

- Instruction Set Architecture (ISA)
- Assembly and machine language

Where we've been: **abstractions**

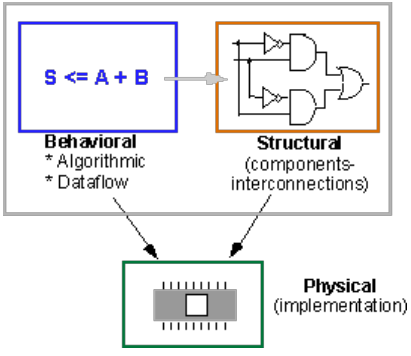
- Instruction Set Architecture (ISA)
- Assembly and machine language

What's up ahead: Implementing the ALU architecture



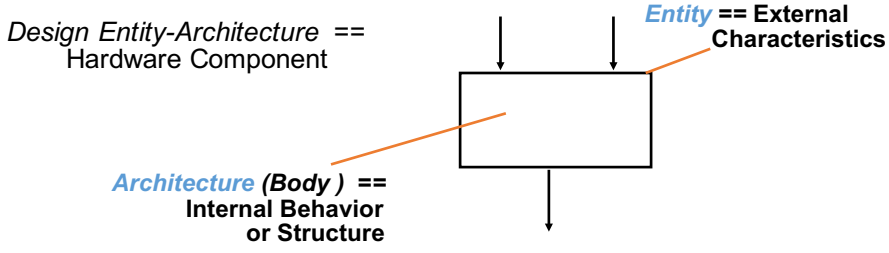
Review: VHDL

- Supports design, documentation, simulation & verification, and synthesis of hardware
- Allows integrated design at behavioral & structural levels



Basic Structure

- Design entity-architecture descriptions
- Time-based execution (discrete event simulation) model



Review: Entity-Architecture Features

Entity

defines externally visible characteristics

- Ports: channels of communication
  - signal names for inputs, outputs, clocks, control
- Generic parameters: define class of components
  - timing characteristics, size (fan-in), fan-out

Review: Entity-Architecture Features (cont.)

Architecture

defines the internal behavior or structure of the circuit

- Declaration of internal signals
- Description of behavior
  - collection of Concurrent Signal Assignment (CSA) statements (indicated by <=);
  - can also model temporal behavior with the delay annotation
  - one or more processes containing CSAs and (sequential) variable assignment statements (indicated by :=)
- Description of structure
  - interconnections of components; underlying behavioral models of each component must be specified

ALU VHDL Representation

```
entity ALU is
  port(A, B:  in std_logic_vector (31 downto 0);
          m:  in std_logic_vector (3 downto 0);
          result: out std_logic_vector (31 downto 0);
          zero: out std_logic;
          ovf: out std_logic)
end ALU;

architecture process_behavior of ALU is
. . .
begin
    ALU: process(A, B, m)
    begin
        . . .
        result := A + B;
        . . .
    end process ALU;
end process_behavior;
```

Machine Number Representation

- Bits are just bits (have no inherent meaning)[1]
- Binary numbers (base 2) – integers

Of course, it gets more complicated:

- storage locations (e.g., register file words) are finite, so have to worry about overflow (i.e., when the number is too big to fit into 32 bits)
- have to be able to represent negative numbers, e.g., how do we specify -8 in

```
addi    $sp, $sp, -8    #$sp = $sp - 8
```

- in real systems have to provide for more than just integers, e.g., fractions and real numbers (and floating point) and alphanumeric (characters)

[1] conventions define the relationships between bits and numbers

RISC-V Representation

32-bit signed numbers (2's complement):

$$
\begin{aligned}
&0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}\\
&0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = +\ 1_{ten}\\
&0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = +\ 2_{ten}\\
&\ldots\\
&0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = +\ 2{,}147{,}483{,}646_{ten}\\
&0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = +\ 2{,}147{,}483{,}647_{ten}\\
&1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -\ 2{,}147{,}483{,}648_{ten}\\
&1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -\ 2{,}147{,}483{,}647_{ten}\\
&1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -\ 2{,}147{,}483{,}646_{ten}\\
&\ldots\\
&1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -\ 3_{ten}\\
&1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -\ 2_{ten}\\
&1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -\ 1_{ten}
\end{aligned}
$$

What if the bit string represented addresses?

- need operations that also deal with only positive (unsigned) integers

Two's Complement Operations

- Negating a two's complement number – complement all the bits and then add a 1
  - remember: "negate" and "invert" are quite different!

- Converting n-bit numbers into numbers with more than n bits:
  - MIPS 16-bit immediate gets converted to 32 bits for arithmetic
  - sign extend: copy the most significant bit (the sign bit) into the other bits

```
0010  -> 0000 0010
1010  -> 1111 1010
```

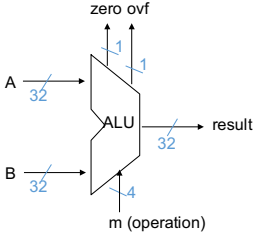
  - sign extension versus zero extend (`lb` vs. `lbu`)

Design the RISC-V Arithmetic Logic Unit (ALU)

- Must support the Arithmetic/Logic operations of the ISA

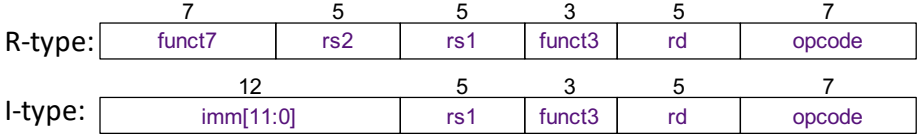
```
RV 32I:
add, sub, mul, mulh, mulhu, mulhsu,
div, divu, rem, li, addi, sll, srl,
sra, or, xor, not, slt, sltu, slli,
srli, srai, andi, ori, xori, slti,
sltiu,

RV 64I:
addw, subw, remu, mulw, divw, divuw,
remw, remuw, addiw, sllw, srlw, sraw,
srliw, sraiw,
```



- With special handling for:
  - sign extend: `addi, slti, sltiu`
  - zero extend: `andi, xori`
  - Overflow detected: `add, addi, sub`

RISC-V Arithmetic and Logic Instructions



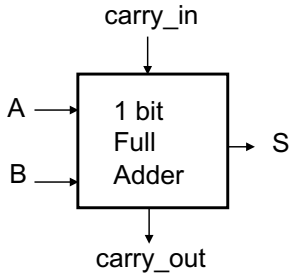
Type	op	funct
ADDI	001000	xx
ADDIU	001001	xx
SLTI	001010	xx
SLTIU	001011	xx
ANDI	001100	xx
ORI	001101	xx
XORI	001110	xx
LUI	001111	xx

Type	op	funct
ADD	000000	100000
ADDU	000000	100001
SUB	000000	100010
SUBU	000000	100011
AND	000000	100100
OR	000000	100101
XOR	000000	100110
NOR	000000	100111

Type	op	funct
	000000	101000
	000000	101001
SLT	000000	101010
SLTU	000000	101011
	000000	101100

Addition Unit

Building a 1-bit Binary Adder

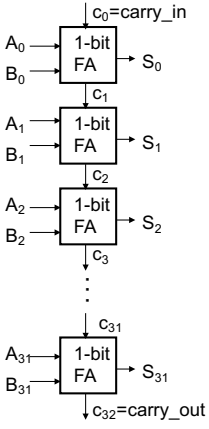


A	B	carry_in	carry_out	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

S = A xor B xor carry_in
carry_out = A&B | A&carry_in | B&carry_in
(majority function)

- How can we use it to build a 32-bit adder?
- How can we modify it easily to build an adder/subtractor?

Building 32-bit Adder

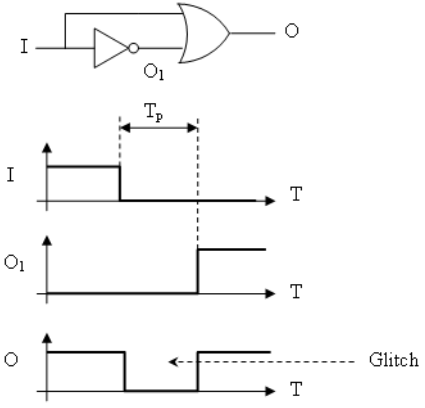


- Just connect the carry-out of the least significant bit FA to the carry-in of the next least significant bit and connect ...
- Ripple Carry Adder (RCA)
  - ☺: simple logic, so small (low cost)
  - ☹: slow and lots of glitching (so lots of energy consumption)

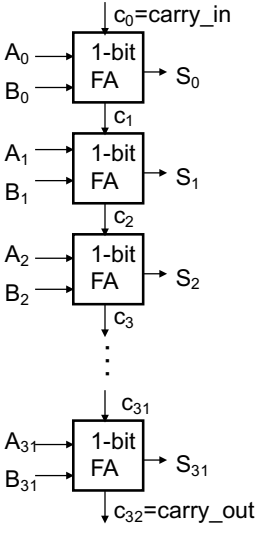
Glitch

invalid and unpredicted output that can be read by the next stage and result in a wrong action

Example: Draw the propagation delay



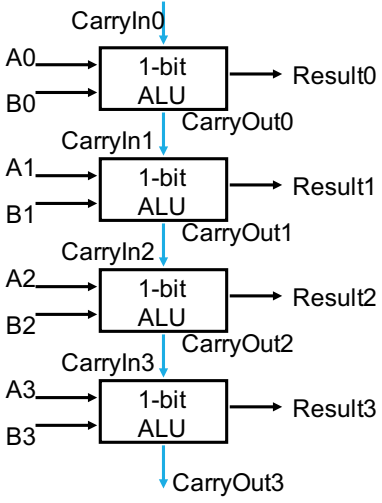
Glitch in RCA



A	B	carry_in	carry_out	S
0	0	**0**	0	**0**
0	0	**1**	0	**1**
0	1	**0**	0	**1**
0	1	**1**	1	**0**
1	0	**0**	0	**1**
1	0	**1**	1	**0**
1	1	**0**	1	**0**
1	1	**1**	1	**1**

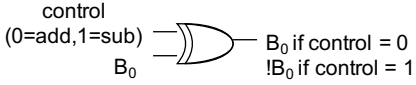
But What about Performance?

- Critical path of n-bit ripple-carry adder is $n \times CP$
- Design trick: throw hardware at it (Carry Lookahead)



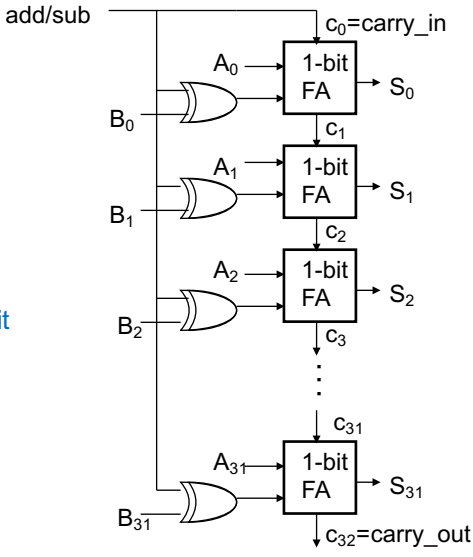
A 32-bit Ripple Carry Adder/Subtractor

- complement all the bits



- add a 1 in the least significant bit

```
A    0111   ->    0111
B  - 0110   -> +  1001
     0001              1
                  1 0001
```



Minimal Implementation of a Full Adder

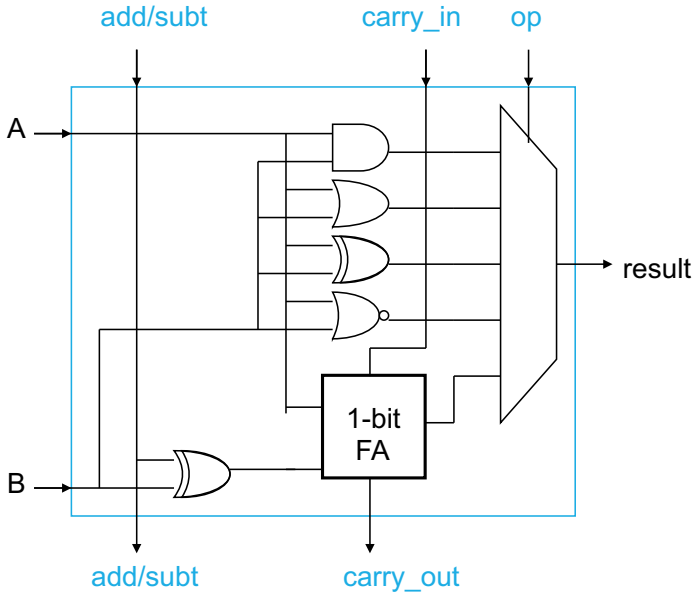
Gate library: inverters, 2-input NANDs, or-and-inverters

```
architecture concurrent_behavior of full_adder is
   signal t1, t2, t3, t4, t5: std_logic;
begin
     t1 <= not A after 1 ns;
     t2 <= not cin after 1 ns;
     t4 <= not((A or cin) and B) after 2 ns;
     t3 <= not((t1 or t2) and (A or cin)) after 2 ns;
     t5 <= t3 nand B after 2 ns;
     S <= not((B or t3) and t5) after 2 ns;
     cout <= not((t1 or t2) and t4) after 2 ns;
end concurrent_behavior;
```

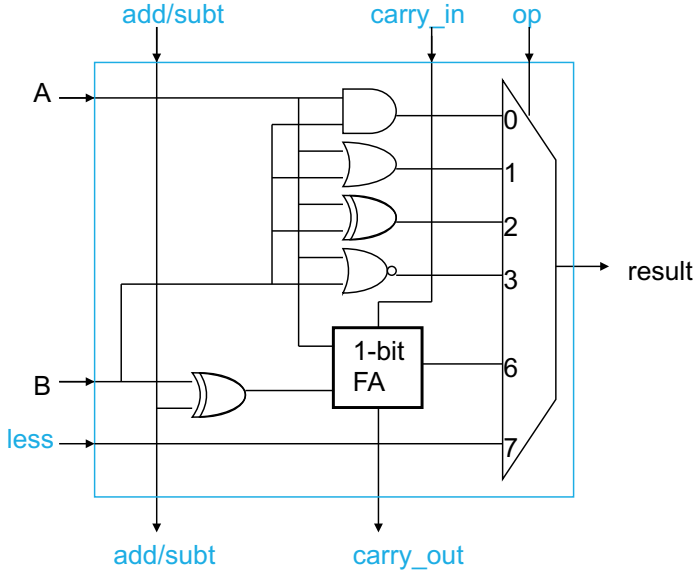
Tailoring the ALU to the MIPS ISA

- Also need to support the logic operations (`and`, `nor`, `or`, `xor`)
  - Bit wise operations (no carry operation involved)
  - Need a logic gate for each function and a mux to choose the output
- Also need to support the set-on-less-than instruction (slt)
  - Uses subtraction to determine if $(a - b) < 0$ (implies $a < b$)
- Also need to support test for equality (`bne`, `beq`)
  - Again use subtraction: $(a - b) = 0$ implies $a = b$
- Also need to add overflow detection hardware
  - overflow detection enabled only for `add, addi, sub`
- Immediates are sign extended outside the ALU with wiring (i.e., no logic needed)

A Simple ALU Cell with Logic Op Support



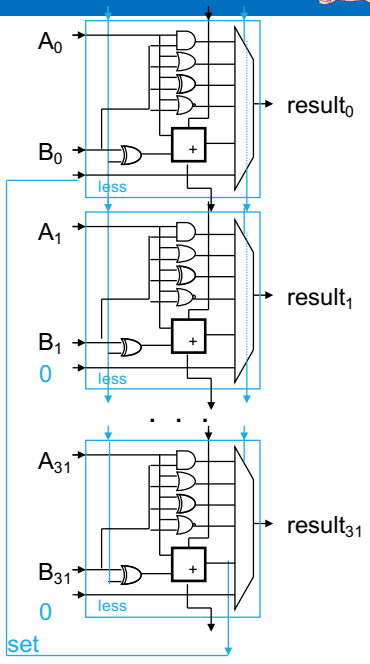
A Simple ALU Cell with Logic Op Support



Modifying the ALU Cell for `slt`

Modifying the ALU for `slt`

- First perform a subtraction
- Make the result 1 if the subtraction yields a negative result
- Make the result 0 if the subtraction yields a positive result
- Tie the most significant sum bit (sign bit) to the low order less input



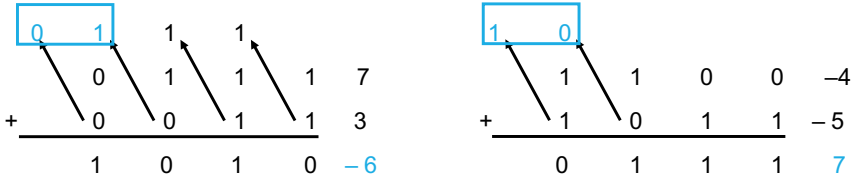
Overflow Detection

Overflow occurs when the result is too large to represent in the number of bits allocated

- adding two positives yields a negative
- or, adding two negatives gives a positive
- or, subtract a negative from a positive gives a negative
- or, subtract a positive from a negative gives a positive

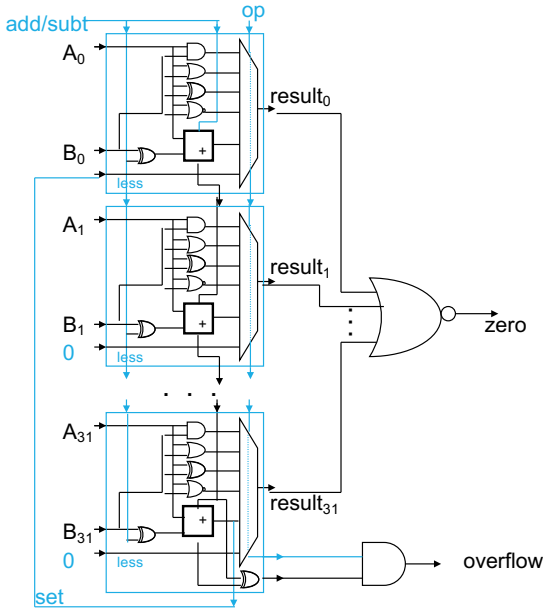
Question: **prove** you can detect overflow by:

Carry into MSB xor Carry out of MSB



Modifying the ALU for Overflow

- Modify the most significant cell to determine overflow output setting
- Enable overflow bit setting for signed arithmetic (`add`, `addi`, `sub`)



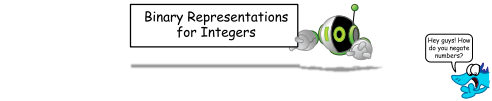
Overflow Detection and Effects

- On overflow, an exception (interrupt) occurs
- Control jumps to predefined address for exception
- Interrupted address (address of instruction causing the overflow) is saved for possible resumption
- Don't always want to detect (interrupt on) overflow

New MIPS Instructions

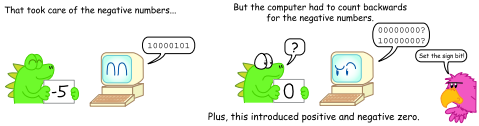
Category	Instr	Op Code	Example	Meaning
Arithmetic (R & I format)	add unsigned	0 and 21	addu $s1, $s2, $s3	$s1 = $s2 + $s3
	sub unsigned	0 and 23	subu $s1, $s2, $s3	$s1 = $s2 - $s3
	add imm.unsigned	9	addiu $s1, $s2, 6	$s1 = $s2 + 6
Data Transfer	ld byte unsigned	24	lbu $s1, 20($s2)	$s1 = Mem($s2+20)
	ld half unsigned	25	lhu $s1, 20($s2)	$s1 = Mem($s2+20)
Cond. Branch (I & R format)	set on less than unsigned	0 and 2b	sltu $s1, $s2, $s3	if ($s2<$s3) $s1=1 else $s1=0
	set on less than imm unsigned	b	sltiu $s1, $s2, 6	if ($s2<6) $s1=1 else $s1=0

- Sign extend: `addi, addiu, slti`
- Zero extend: `andi, ori, xori`
- Overflow detected: `add, addi, sub`



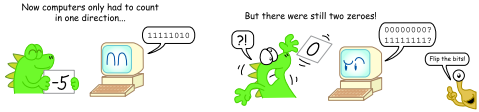
In the early days of computing, designers made computers express numbers using **unsigned binary**.

To include negative numbers, designers came up with **sign magnitude**.



Plus, this introduced positive and negative zero.

Then designers created **one's complement**.



Finally, designers developed **two's complement**.